CS3230 Cheatsheet

github.com/reidenong/cheatsheets, AY25/26 Sem 1.

Asymptotic Analysis

if $\exists c, c_1, c_2, n_0 > 0$ s.t. $\forall n \geq n_0$,

$f(n) \in O(g(n))$	$0 \leq f(n) \leq c \cdot g(n)$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in \Omega(g(n))$	$0 \leq c \cdot g(n) \leq f(n)$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) \in \Theta(g(n))$	$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

if $\forall c > 0, \exists n_0 > 0$ s.t. $\forall n \geq n_0$,

$f(n) \in o(g(n))$	$0 \leq f(n) < c \cdot g(n)$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n))$	$0 \leq c \cdot g(n) < f(n)$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Master Theorem

Given $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, where $f(n) = c \cdot n^m \log^k n$, Let $d = \log_b a$.
Work to split/recombine at each root is $f(n)$.
Work at the leaves is # of leaves, given by n^d.

Case 1: $f(n) \in O(n^{d-\varepsilon})$	$T(n) \in \Theta(n^d)$ work at leaves dominate.
Case 2a: $f(n) \in \Theta(n^d \log^k n)$, $k > -1$	$T(n) \in \Theta(n^d \log^{k+1} n)$
Case 2b: $f(n) \in \Theta(n^d \log^k n)$, $k = -1$	$T(n) \in \Theta(n^d \log\log n)$
Case 2c: $f(n) \in \Theta(n^d \log^k n)$, $k < -1$	$T(n) \in \Theta(n^d)$
Case 3*: $f(n) \in \Omega(n^{d+\varepsilon})$, assuming $\exists c < 1$, s.t. $\forall x$, $a \cdot f\left(\frac{x}{b}\right) \leq c \cdot f(x)$	$T(n) \in \Theta(f(n))$ work (to split/recombine) at roots dominate.

* if given $f(n)$ is not of the form $f(n) = c \cdot n^d \log^k n$, the regularity condition may not hold, and must be checked.

Proofs of Correctness

Iterative Algorithms Correctness

1. Establish loop invariant
2. Initialization (Inv true before iteration 1)
3. Maintenance (satisfied at start/end of iteration)
4. Termination (Inv true when algo ends)

Dijkstra

- Invariant
  - $R \subseteq V$ is the set of *settled* vertices.
  - $\forall u \in R, d(u) = \text{dist}(s, u)$
- Selection rule in iteration
  - Pick vertex $v \in V \setminus R$ that minimizes $\min_{u \in R}\{d(u) + w(u, v)\}$, sets it as $d(v)$, and adds v to R.

Implementation

	Binary Heap	Fib Heap
Insert	$O(\log n)$	$O(1)$
Extract-min	$O(\log n)$	$^*O(\log n)$
Decrease-key	$O(\log n)$	$^*O(1)$
Total	$O((m+n)\log n)$	$O(m + n\log n)$

* amortized

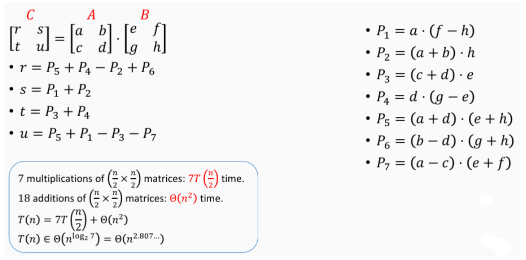
Recursive Algorithms Correctness

- Induction (true on base cases, true by inductive step)

Divide and Conquer

Strassen's Algorithm

Matrix addition $O(n^2)$ is faster than matrix multiplication $O(n^{2+\varepsilon})$.



$$\overset{C}{\begin{bmatrix} r & s \\ t & u \end{bmatrix}} = \overset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \cdot \overset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}$$

- $r = P_5 + P_4 - P_2 + P_6$
- $s = P_1 + P_2$
- $t = P_3 + P_4$
- $u = P_5 + P_1 - P_3 - P_7$

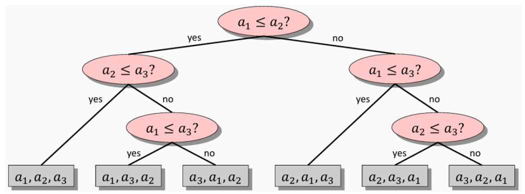
- $P_1 = a \cdot (f - h)$
- $P_2 = (a + b) \cdot h$
- $P_3 = (c + d) \cdot e$
- $P_4 = d \cdot (g - e)$
- $P_5 = (a + d) \cdot (e + h)$
- $P_6 = (b - d) \cdot (g + h)$
- $P_7 = (a - c) \cdot (e + f)$

7 multiplications of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices: $7T\left(\frac{n}{2}\right)$ time.
18 additions of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices: $\Theta(n^2)$ time.
$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$
$T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807\ldots})$

Lower Bound with Decision Trees

Comparison Based Sorting

- Starting at the root, at every inner node, a decision is made. At a leaf, a decision is taken.
- Worst case running time of comparison sorting $\geq$ worst-case number of comparisons = height of decision tree



- Any binary tree of height k has at most 2^k leaves

$\Rightarrow$ With each permutation as a possible answer (leaf), the height of the binary tree is at least $\log(n!)$. By Stirling's approximation,

$$\log(n!) \in n\log n - n\log e + O(\log n) \subseteq \Theta(n \log n)$$

Average Case Analysis

- The average case running time $A(n)$ is the mean runnning time over all inputs of size n.
- $A(n)$ is also the expected running time when the permutation π is chosen uniformly at random.

Quicksort

$$A(n) = \sum_{\pi} \frac{1}{n!} \cdot (\text{running time of quicksort on } \pi)$$

1. Pivot is selected uniformly at random
2. Permutations for both recursive calls are also uniformly random

Using the second definition of average case,

$$\begin{aligned} &A(n) \\ &= \sum_{j \in N} \Pr[\text{pivot} = a_j] \\ &\cdot [\text{Recursive calls left and right} + \text{Work to partition}] \\ &= \frac{1}{n} \sum_{j \in N} [A(j-1) + A(n-j) + cn] \\ &= cn + \frac{2}{n} \sum_{j \in N} A(j) \\ &\in O(n \log n) \end{aligned}$$

'Stooge Sort' (Tutorial 3)

```
Let $n$ be the length of array $A$
if $n = 2$ and $A[0] > A[1]$ then
    Swap $A[0]$ and $A[1]$
if $n > 2$ then
    Apply StoogeSort to sort the first $\lceil 2n/3 \rceil$ elements recursively
    Apply StoogeSort to sort the last $\lceil 2n/3 \rceil$ elements recursively
    Apply StoogeSort to sort the first $\lceil 2n/3 \rceil$ elements recursively
```

Time: $O\left(n^{\log_{\frac{3}{2}} 3}\right) = O(n^{2.7095\ldots})$

Stable Sorting

- For elements of equal values, the original ordering is preserved.

In-place sorting

- Uses very little extra memory beside input array
  - Insertion sort $O(1)$, Quicksort $O(\log n)$, Mergesort $O(n)$

Randomized Algorithms

Matrix Multiplication Verification

Given 3 $n \times n$ matrices A, B, C, check if $AB = C$. MM can be done in $O(n^3)$ naively, or $O(n^{2.8})$ with Strassen's algorithm.

Freivalds Algorithm

1. Choose $v = (v_1, v_2, \ldots, v_n)$ to be a uniformly random column vector from $\{0, 1\}^n$.
2. Check if $ABv = Cv$ in $O(n^2)$ with matrix-vector multiplication.
3. If $ABv = Cv$, then output $AB = C$ (Always correct)
3. If $ABv \neq Cv$, then output $AB \neq C$

False Positive

We focus on the false positive case where $AB \neq C$, but the algorithm outputs otherwise.

$$C^* = AB - C = \begin{pmatrix} c^*_{1,1} & \cdots & c^*_{1,n} \\ \vdots & \ddots & \vdots \\ c^*_{n,1} & \cdots & c^*_{n,n} \end{pmatrix}$$

$$\text{let } u = C^* v = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$$

Then the algorithm is incorrect $\Leftrightarrow u = \mathbf{0}$.

1. Since $AB \neq C$, $C^* \neq 0$
2. Suppose some row i of C^* is non-zero. Then
$$u_i = \sum_j^n (c^*_{i,j})(v_j) \text{ where some } c^*_{i,j} \neq 0$$
3. We used the principle of deferred decisions:
   - Instead of fixing all random bits of $v = (v_1, \ldots, v_n)$, we analyze one coordinate at the moment it matters.
   - We fix all random numbers of v except for one where $c^*_{i,j} \neq 0$. Then at this point,
$$u_i = (\text{fixed term}) + c^*_{i,j}(v_j)$$
   Now, only one choice of v_j makes $u_i = 0$.
4. $\therefore$, $P(\text{Freivalds successful}) \geq \frac{1}{2}$

Success Probability Amplification

We have algorithm A, with $P(A \text{ fails}) \leq k$. Then we can amplify the success probability by repeating the algorithm t times.

$$P(\text{success}) \geq 1 - k^t$$

Union Bound

For events $A_1, ..., A_k$, the probability that at least one of the events happen is at most the sum of their individual probabilities.

$$\Pr[\cup^k A_i] \leq \sum^k \Pr[A_i]$$

Balls and Bins

- Throw m balls into n bins randomly and independently
- What is the probability that every bin contains at least one ball?

1. Consider 1 ball, n bins.

$P(\text{bin}_i \text{ has ball}) = \frac{1}{n}$
$\Rightarrow P(\text{bin}_i \text{ has no ball}) = 1 - \frac{1}{n}$

2. Consider m balls, n bins.

$P(\text{bin}_i \text{ has no ball})$
$= \left(1 - \frac{1}{n}\right)^m$
$= \left(1 - \frac{1}{n}\right)^{n \cdot \left(\frac{m}{n}\right)}$
$\leq e^{-\frac{m}{n}}$
as $1 + x \leq e^x$.

3. By union bound,

$$\begin{aligned} &P(\text{at least one bin is empty}) \\ &\leq n \cdot P(\text{bin}_i \text{ has no ball}) \quad \text{by union bound} \\ &\leq ne^{-\frac{m}{n}} \end{aligned}$$

If $m \geq 2n\lceil \ln n \rceil$,

$$P(\text{at least one bin is empty}) \leq \frac{1}{n}$$

$\therefore$, throwing $m = 2n\lceil \ln n \rceil \in \Theta(n \log n)$ boxes guarantees a success probability of $\geq 1 - \frac{1}{n}$

Markov Inequality

- If X is a non-negative random variable and $a > 0$, then

$$P(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

- Then if the expected runtime of A is at most T, the runtime of A is at most $100 \cdot T$ with probability ≥ 0.99.
  - Expected time complexity of A is $T(n)$, then time complexity of A is $T(n)$ with probability ≥ 0.99

Randomized QuickSort

- From an array $A[1, ..., n]$ of distinct numbers, select a number as the pivot uniformly at random. We then rearrange the array to satisfy the condition, and recursively sort left and right halves.
- Running time of quicksort $\in O(\text{number of comparisons})$

Randomized Quicksort Time Analysis

- Consider $a = (a_1, ..., a_n)$ the numbers of A in sorted order.
- The number of comparisons made between 2 elements $a_i, a_j, i < j$ is either 0 or 1, and this occurs if either one becomes the pivot.
- Before a number in $a[i : j]$ is selected, the numbers in $a[i : j]$ must belong to the same array
  - If a_i or a_j are the first numbers chosen as a pivot in $a[i : j]$, the comparison between a_i, a_j happens.
  - Otherwise, a_i, a_j will belong to different arrays in the recursive calls and no comparison occurs
  - Hence a comparison is made between $a_i, a_j \Leftrightarrow$ the first number chosen as a pivot in $a[i : j]$ is either of them
- $\therefore, i < j, P(\text{comparison}_{i,j}) = \frac{2}{j-i+1}$.

$$\begin{aligned} &\mathbb{E}[\text{Number of comparisons}] \\ &= \sum_{i<j}^{n} \frac{2}{j-i+1} \\ &= 2\sum_{i}^{n} \sum_{j=i}^{n} \frac{1}{j-i+1} \\ &= 2\sum_{i}^{n} \left(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n-i+1}\right) \\ &\in O(n \log n) \end{aligned}$$

Types of Randomized Algorithms

- Las Vegas algorithm
  - Output always correct
  - Time complexity guarantee is in expectation
  - Randomized Quicksort
- Monte Carlo algorithm
  - Output correct with some probability
  - Time complexity guarantee is in expectation
  - Freivald's algorithm
- Las Vegas Algorithms can be turned into Monte Carlo via Markov Inequality
  - We use Markov inequality to bound the probability that runtime is larger than expected, then cut the algorithm if it runs too long.

Greedy Algorithms

A problem that admits a greedy strategy relies on both the fact that the greedy choice is safe as well the optimal substructure property.

After making a safe greedy choice, given the optimal substructure property, we end up with another (sub)instance of the same problem to which we can reapply the greedy choice.

1. Redefine the problem s.t. we make a greedy choice and are left with a subproblem to solve
2. Prove via exchange argument that there is always an optimal solution that makes the greedy choice, so it is safe
3. Use optimal substructure to show we can combine an optimal solution with the greedy choice

Fractional Knapsack

Greedy Choice

Let j^* be the item with maximum value/weight. Then there exists an optimal knapsack containing $\min(w_{j^*}, W)$ kg of item j.

Proof by exchange argument:

Suppose an optimal knapsack has $x_{j^*} < \min(w_{j^*}, W)$.

From the rest of the knapsack, pick y_i of item i, such that the weights add up to $\min(w_{j^*}, W) - x_{j^*}$. We can then replace these weights by $\min(w_{j^*}, W) - \sum y_{j^*}$, total weight does not change, and total value does not decrease as j^* is the item with maximum value/weight.

The knapsack stays optimal and it is safe to use the greedy choice.

Optimal substructure

If we remove y kg of an item j from the *optimal* knapsack, then the remaining load must be the *optimal* knapsack weighing at most $W - y$ kg that one can take from the other $n - 1$ items and $w_j - y$ kg of item j.

Proof by cut and paste:
Let X be the value of the optimal knapsack.

Suppose that the remaining load after removing y kg of item j was not the optimal knapsack weighing at most $W - y$ kg that one can take from the $n - 1$ other items and $w_j - y$ kg of item j

This means that there is other knapsack of value $> X - v_j \cdot \left(\frac{y}{w_j}\right)$ with weight $\leq W - y$ kg among the $n - 1$ other items and $-y$ kg of item j. If we combine this other knapsack with y kg of item j, we will have a 'more optimal knapsack' of value $> X$ with weight $\leq W$, which is a contradiction.

$\therefore$, the optimal substructure must be optimal

Huffman Encoding

When encoding an alphabet into a binary sequence, we can exploit variations in alphabet frequency to use a variable length encoding where higher frequency alphabets have shorter codes, leading to a overall shorter bit length.

Theorem

For each prefix code of a set A on n alphabets, there exists a binary tree T on n leaves such that

- There is a 1-1 mapping between the alphabets and leaves
- The label of a path from root to a leaf corresponds to the prefix code of the corresponging alphabet at a leaf.

Lemma

There exists a optimal prefix coding in which a_1, a_2 (alphabet sorted in non decreasing frequencies) appear as siblings in the corresponding labelled binary tree.

Amortized Analysis

In a sequence of operations it can be shown that the average cost per operation is small, despite having a few expensive operations. We can use this to get a guarantee on the average performance of each operation in the worst case.

Aggregate Method

We compute the total cost of a sequence of n operations, then divide by n to get the amortized cost per operation. This works best when all operations are similar.

Example: Binary counter

Each bit flips about half as often as the one before it, so

$$\text{total flips} = \left(n + \frac{n}{2} + \frac{n}{4} + ...\right) = O(n)$$

Amortized cost of *increment* is thus $O(1)$.

Accounting Method

We can assign each operation an amortized charge. We overcharge cheap operations and use it to pay for future expensive ones. It is important that the balance never goes below 0.

Example Binary counter:

Charge \$2 for each set$(0 \to 1)$, \$1 for itself and \$ as surplus, and later when resetting$(1 \to 0)$ we can use the surplus to pay for itself. This is true as after i increments, the amount of money in the bank is also the number of on bits in the binary representation of i.

Potential Method

Choose potential function ϕ as something that decreases a lot after the expensive operations.

For each operation, we then need 3 things

- Actual Cost, $T(i)$
- Potential difference, $\phi(i) - \phi(i-1)$
- Amortized cost,$C(i) = T(i) + \Delta\phi(i)$

such that $\forall i > 0, \phi(i) \geq 0.$ and $\phi(0) = 0$

Let ϕ be the potential function associated with the algorithm. If ϕ increases, extra work is being saved up, and if ϕ decreases, saved work is being spent. Then $\phi(i)$ is the potential at the end of the i-th operation.

- $\phi(0) = 0$
- $\forall i, \phi(i) \geq 0$

Then we have

$$\begin{aligned} &\text{Amortized cost of i-th operation} \\ &= \text{actual cost of i-th operation} + \phi(i) - \phi(i-1) \\ &= \text{actual cost of i-th operation} + \Delta\phi(i) \end{aligned}$$

And in turn,

$$\begin{aligned} &\text{Amortized cost of n operations} \\ &= \sum_i \text{amortized cost of i-th operation} \\ &= \text{actual cost of n operations} + (\phi(n) - \phi(0)) \\ &\text{by telescoping} \\ &= \text{actual cost of n operations} + \phi(n) \\ &\geq \text{actual cost of n operations} \end{aligned}$$

Then to show the actual cost of n operations is $\in O(g(n))$, just show the amortized cost of n operations is $\in O(g(n))$.

Example: Dynamic array

Insertion of an element into a dynamic array that doubles when full: most inserts cost 1, and occasionally, resizing costs n.

We can define $\phi(i) = 2n - \text{capacity}$.

Case 1: Regular Insertion
capacity is unchanged, ϕ increases by 2. Amortized cost $= 3 \in O(1)$

Case 2: Resizing

$$\phi(i) = 2i - \text{new size} = 2i - 2(i-1) = 2$$
$$\phi(i-1) = 2(i-1) - \text{old size} = 2(i-1) - (i-1) = i-1$$
$$\Delta\phi(i) = 3 - i$$

Actual cost = i, $\Delta\phi(i) = 3 - i$, amortized cost $= 3 \in O(1)$

Problem Reductions

If there is an $O(n^c)$-time reduction from A to B for some constant c, we say there is a polynomial-time reduction from A to B, ie.

$$A \underset{P}{\leq} B$$

- Informally, A is not harder than B
- A is a special case of B
- To prove this:
  1. Define the transformer f, with an algorithm that maps any instance $x \in A$ to a instance $y = f(x) \in B$.
  2. Prove correctness in two directions, $x \in A \to f(x) \in B$ and $x \notin A \to f(x) \notin B$.
  3. Prove time bound, that f runs in time polynomial to $|x|$.

Polynomial time

- Closed under compositions, if $T(n)$, $P(n)$ polynomial, then $T(P(n))$ is also polynomial
- Polynomial time notion is robust under different computing models and hardware
- When and algorithm is polynomial, we mean that the runtime is polynomial relative to the length of the encoding of the problem instance in terms of number of bits
  - Many numeric problems are not numerical, since we can represent the input in $O(\log n)$ (in decimal or binary encoding) and the runtime can be linear: this becomes exponential in runtime.

Pseudo Polynomial time

- An algorithm that runs in time polynomial in the numeric value of the input.
  - Could be exponential in the length of the input encoding

Decision problems

- A decision problem is a function that maps an instance space I to the solution space $\{\text{YES}, \text{NO}\}$.
- Any optimization problem can be converted into a decision problem:
  - Given an instance of the optimization problem and a number k, decide if there exists a solution whose value is $\leq k$ for minimization (or $\geq k$ for maximizatiion)
  - The decision problem reduces to optimization, ie. no harder than the optimization problem.
- Optimization problems can also be reduced to decision problems
  - Perform binary search using the decision solver for the optimal solution
- $\therefore$ the decision problem can be solved in polynomial time $\Leftrightarrow$ the optimization problem can be solved in polynomial time.

VC vs IS

- $X \subseteq V$ is a vertex cover $\Leftrightarrow$ $V \setminus X$ is an independent set.

Proof:
($\rightarrow$):
- Consider $u, v \in V \setminus X$, we just need to show $\{u, v\} \notin E$
- If $\{u, v\} \in E$, then it is an edge not covered by X, which is impossible.

($\leftarrow$):
- We need to show that $\forall \{u, v\} \in E$ at least one of u, v is in X.
- If both u, v not in X, then $V \setminus X$ is not an independent set.

- Then VC $\underset{P}{\leq}$ IS, and there is a transformation $(G, k) \to (G, |V| - k)$.
  - The symmetrical case is true.

NP Completeness

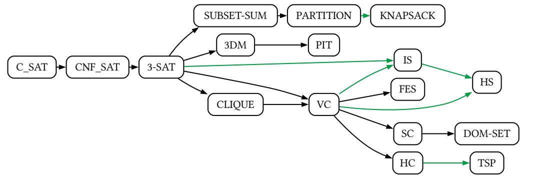
- A verifier is an algorithm A which takes an instance I of a decision problem, a *certificate* s and outputs YES / NO, verifying whether the certificate is right or wrong.
  - A must run in polynomial time to be efficient
  - s must be short, ie. $|s| \leq p(|I|)$ where p is a polynomial function
- **Non-deterministic Polynomial Time (NP)**: The set of all decision problems with an efficient verifier
- **Polynomial Time (P)**: The set of all decisiion problems with an efficinet algorithm
- A problem X is in **NP-Hard** if for every $A \in$ NP, $A \underset{P}{\leq} X$ (X is at least as hard as any other problem in NP)
- A problem X is in **NP-complete** if X is at least as hard as any other problem in NP and is also in NP.

Showing NP-completeness

There are two steps to show a problem X is NP-complete

1. Show that $X \in$ NP.
2. Pick a problem A that is NP-complete, and then show that $A \underset{P}{\leq} X$ (X is NP-hard).

Reduction Tree and Common NP-complete problems



$A \to B$: A reduces to B / $A \underset{P}{\leq} B$

Circuit Satisfiability

- Instance: A boolean combinatorial circuit C with AND, OR, NOT gates and designated output.
- Question: Does there exist an input assignment to the circuit's input such that the output evalutes to True?

CNF Satisfiability

- Instance: A boolean formula in CNF.
- Question: Does there exist a truth assignment to variables such that the formula is true?
- Conjunctive Normal Form: A formula that is a conjunction (AND) of clauses, which are a disjunction (OR) of literals.

3-SAT

- Instance: A CNF boolean formula where each clause has exactly 3 literals.
- Question: Is this formula satisfiable?

Vertex Cover

- Instance: Graph $G = (V, E)$, integer k
- Question: Is there a set $S \subseteq V, |S| \leq k$ such that every edge in E has at least one endpoint in S?
- Minimal vertex cover
- $S \subseteq V$ is an independent set $\Leftrightarrow V \setminus S$ is a vertex cover

Independent set

- Instance: Graph $G = (V, E)$, integer k
- Question: Is there a set $S \subseteq V, |s| \geq k$ such that no two vertices in S share an edge?
- Maximal Independent set

Clique

- Instance: Graph $G = (V, E)$, integer k
- Question: Is there a set $S \subseteq V, |S| \geq k$ such that every pair of vertices in S is adjacent?

Hamiltonian Cycle

- Instance: Graph $G = (V, E)$
- Question: Does G contain a simple cycle that visits every vertex at exactly once?

Travelling Salesman

- Instance: *Weighted* graph $G = (V, E)$, integer W
- Question: Does there exist a tour visitng each vertex exactly once with total weight $\leq W$?

Set Cover

- Instance: Universe $U = \{u_1, ..., u_n\}$, collection of subsets $T = \{S_1, ..., S_m\} \subseteq 2^U$, integer k
- Question: Is there a subcollection of at most k sets whose union equals U?

Feedback Edge Set

- Instance: Undirected Graph $G = (V, E)$, integer k
- Question: Does there exist a set $F \subseteq E, |F| \leq k$ whose removal makes G acyclic?

Hitting Set

- Instance: Universe $U = \{u_1, ..., u_n\}$, collection of subsets $T = \{S_1, ..., S_m\} \subseteq 2^U$, integer k
- Does there exist a set $H \subseteq U, |H| \leq k$ such that $H \cap S_i \neq \emptyset$ for every i?
- Ie. Given a bunch of sets, choose a small set of elements that 'hits' every one

DOM-Set

- Instance: Graph $G = (V, E)$, integer k
- Question: Is there a set $D \subseteq V, |D| \leq k$ such that each vertex is either in D or adjacent to a vertex in D?

3D Matching

- Instance: 3 disjoint sets $X, Y, Z, |X| = |Y| = |Z| = n$, set $T \subseteq X \times Y \times Z$, integer k
- Question: Is there a subset $M \subseteq T, |M| = k$ such that no two triples in M share an element

Partition into Triangles

- Instance: Graph $G = (V, E)$, where $|V| = 3k$ for some k.
- Question: Can the vertices be partitioned into groups of 3 such that each group forms a triangle?

Subset Sum

- Instance: Multiset $S = \{a_1, ..., a_n\}$ of positive integers, integer t
- Question: Does there exist a subset $S' \subseteq S$ whose sum equals t?

Partition

- Instance: Multiset S of positive integers
- Question: Can S be partitioned into two subsets of equal total sum?

0-1 Knapsack

- Instance: Items with weight, value (w_i, v_i), capacity W, target t
- Question: Is there a subset of items with total weight $\leq W$ and total value $\geq t$?

Linear time Sorting & Selection

Counting Sort

- Two pass; the first counts frequencies into DAT, the second outputs in order of frequency
- For stability, get frequency, then calculate starting index of each element with prefix sum. Finally one pass through original array to restore items to index.
- $O(n + k)$, where elements are in $[0, k]$

Radix Sort

- Sort digit-by-digit, LSB to MSB using Counting sort
- for d digits, Radix sort runs d iterations of $O(n + k)$.
- With custom base we can have more optimal runs:
  - split b bit word into $\frac{b}{r}$ groups of r bit words
  - Only $d = \frac{b}{r}$ passes, each pass takes time $O(n + 2^r)$
  - $O(\frac{b}{r} \cdot (n + 2^r))$
  - Optimal is when $r = \log n$ as $n + 2^r$ balances
  - Total time is $O\left(\frac{b \cdot n}{\log n}\right)$
  - $b = d \cdot \log n$ bits $\Rightarrow$ Radix sort runs in $O(d \cdot n)$

Order Statistics

- Given an unsorted array without duplicates, find the index of the rank i element

Quickselect (Randomized)

- DnC, similar to randomized quicksort but only recurse on half containing i.

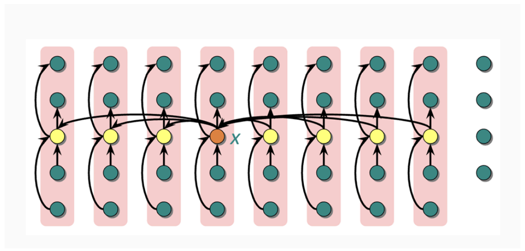
```
QuickSelect($A, l, r, i$)
    if $l == r$: return $l$
    $k$ = RandomizedPartition($A, l, r$)
    if $i == k$: return $k$
    else if $i < k$: return QuickSelect($A, l, k-1, i$)
    else: return QuickSelect($A, k+1, r, i$)
```

- Best case: $O(n)$, Worst case: $O(n^2)$

Deterministic linear time selection algorithm

```
Select($A, n, i$)
    divide $A$ into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements each
    for each group in $A$:
        find the median of the 5-element group in $O(1)$
    Let $B$ be the array of medians (size $\lfloor \frac{n}{5} \rfloor$)
    $x = B[\text{Select}(B, |B|, \frac{|B|}{2})]$
    k = Partition($A, x$)
    Let $A'$ be the elements of A with value $< x$
    Let $A''$ be the elements of A with value $> x$
    if $i == k$: return $k$
    else if $i < k$: return Select($A', k-1, i$)
    else: return Select($A'', n-k, i-k$)
```

Finding Median of medians



- Arrange all elements in groups of 5, find their median. Then recurse on these until we have the median or medians.
- At least half of the group of 5 medians are $\leq x$, meaning at least $\lfloor \frac{n}{10} \rfloor$ medians. $\therefore$, at least $3\lfloor \frac{n}{10} \rfloor$ elements are $\leq x$.
- By symmetry, at least $3\lfloor \frac{n}{10} \rfloor$ elements are $\geq x$.
- We recurse on at most $\frac{7n}{10}$ elements in the worst case

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n) \\ &\leq c \cdot \frac{n}{5} + c \cdot \frac{7n}{10} + \Theta(n) \\ &\leq c \cdot \frac{9n}{10} + \Theta(n) \\ &\leq c \cdot n - \left(c \cdot \frac{n}{10} - d \cdot n\right) \\ &\leq c \cdot n \end{aligned}$$

Dynamic Order Statistics

- BBST with size-of-tree augmentations for $O(\log n)$ select and rank.
- Fenwick Tree, Segment Tree solutions with binary search on frequency

Miscellaneous

Math Identities

Logarithms

$$a^{\log b} = b^{\log a}$$

$$\log(n!) \in \Theta(n \log n) \text{ by Stirling's approximation}$$

You can take the square root of n about $O(\log \log n)$ times.

$$\sum_i^{\log n} 2^i = O(n), \sum_i^{\log n} \frac{1}{2^i} = O(1),$$

Series

Arithmetic series

$$\sum_k^n k^p \in \Theta(n^{p+1})$$

Geometric series

$$\sum_k^n x^k = 1 + x + x^2 + ... + x^n = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_k^\infty x^k = \frac{1}{1 - x} \text{ when } |x| < 1$$

Harmonic series

$$\sum_k^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} = \ln n + O(1)$$

Limits

L'Hopital's Rule

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$$

Probability Identities

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

$$\Pr[A \mid B] = \frac{\Pr[A] \cdot \Pr[B \mid A]}{\Pr[B]}$$

Bernoulli Trials

Suppose we have a sequence of independent Bernoulli trials, each with probability p of success. Let X be the number of trials needed to obtain a success for the first time. Then

$$\Pr[X = k] = (1 - p)^{k-1} p$$

$$E(X) = \frac{1}{p}$$

Suppose X is the number of successes in n Bernoulli trials. Then

$$\Pr[X = k] = \binom{n}{k} p^k \cdot (1 - p)^{n-k}$$

Some other Algorithms

Karatsuba

A DnC algorithm for multiplying two large numbers. Given two numbers

$$\begin{aligned} A &= a_1 \cdot B + a_0 \\ B &= b_1 \cdot B + b_0 \end{aligned}$$

Naively we take 4 multiplications of size $\frac{n}{2}$:

$$A \cdot B = a_1 b_1 \cdot B^2 + (a_1 b_0 + a_0 b_1) \cdot B + a_0 b_0$$

With Karatsuba, observe

$$\begin{aligned} p_0 &= a_0 b_0 \\ p_1 &= a_1 b_1 \\ p_2 &= (a_0 + a_1)(b_0 + b_1) \end{aligned}$$

$$A \cdot B = p_1 \cdot B^2 + (p_2 - p_1 - p_0) \cdot B + p_0$$

With 3 multiplications, we get

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + O(n) \\ &= O(n^{\log_2 3}) = O(n^{1.585...}) \end{aligned}$$

Kruskal

Find a MST using greedy edge picking and UFDS, $O(E \log E)$

Prim

Find a MST in by growing a tree from a single node, greedily picking lightest edges, $O(E \log V)$.